will probably be safest. If your code needs to move quantities of characters between buffers, possibly translating in the process, you may want to consider using hardware instructions to do the job.

Be aware, though, that using either byte instructions (1110 or 1100/80) or the bit-move instruction (1100/60/70 with EIS, 1100/90, or 2200) makes your code dependent on that hardware. (A 2200 operating in extended mode adds still more character instructions.)

If your code has to be transportable, you may have to do the same job three times: once without character instructions, and once each with the above-named instruction sets. A configuration tag could choose from alternate definitions of the same character-move procs. Or, for a maximum in flexibility, you can use the bit settings in A2 and A3 at the start of execution (see the next chapter) to determine the string hardware implementation your program is executing with.

Here is a very basic example using the bit-move instruction.

```
$(0)        AXR$                    . DEFINE REGISTERS
            ASCII                   .
FROMBUF     'THIS IS A TEST'        . SOURCE BUFFER
FROMLEN     EQU     ($-FROMBUF)*36  . BIT LENGTH OF SOURCE
TOBUFF      RES     5               . DESTINATION BUFFER
$(1)        .
START.
            L,U     A0,TOBUFF       . SET UP 'TO' REGISTER
            L       A1,(FROMBUF+1*/17) . SET UP 'FROM' REG
            LXI,U   A1,FROMLEN*/4   . BIT COUNT FIELD
            BIM     A0              . DO THE MOVE
            L$SNAP  'BITMOV',2,15,FROMBUF . SNAP RESULTS
            ER      EXIT$           . QUIT
            END     START           .
```

This program will only run correctly on an 1100/60 or /70 with EIS, an 1100/90, or a 2200. The example actually ends up moving four whole words, but you can experiment with the bit count (FROMLEN) and other instruction parameters yourself.

The main possible confusion factor in this example is the order of parameter words. Like the block transfer instruction's fields, they are backwards from intuition; the "to" parameter is first, and the "from" parameter is second. The example omits the use of index registers and the bit offsets. The parameter words can be in memory or in registers, like the example. The operands (source and destination) are always in memory, never registers.

(You have probably noticed that a FORM directive could be used to create the various fields of the parameters, but that would be less sporting. In practice, the code will probably have to set up the parameters dynamically, the way this example does.) The "h" field (1*/17 on line 9 in this example), if set to zero, can be used to blank- or zero-fill a buffer from a single-word "from" operand. (Deciphering of the instruction parameters in the processor and storage references is left as an exercise for the reader!)

Manipulating characters can be one of the more intricate jobs to do in MASM, but also one of the most fun. Try coding up a test program using one of the character vectors, and move some data around. Using these techniques, you'll have it mastered in no time!

21.5 Converting Fieldata or ASCII To Binary

Many times, your programs may need to accept specification fields from outside, via READ$, COM$, or other ERs. Those inputs will usually be in the form of character strings. If your program will use them to control counting or similar operations, it will have to convert them to binary. And that conversion is easy, using these steps:

- Initialize an accumulator register to zero. This register will eventually hold the result.
- For each character, remove the bias, the arithmetic difference between the character value and the decimal digit. Multiply the accumulator by 10 and add the value of the new digit.
- Stop processing characters when a blank or other terminating character is found.

Here is a sample subroutine to convert a Fieldata string of decimal digits into its binary equivalent. It assumes that the Fieldata number is left justified, space filled, in A0. It returns the binary equivalent in A2. (By the way, this is a prime case of a function that should be split out as an internal subroutine, even if it is called only once.)

```
CVTFD       . CONVERT FD STRING IN A0 INTO BINARY IN A2
            L,U         A1,5        . MAX CHARACTERS MINUS 1
            L,U         A2,0        . INITIALIZE ACCUMULATOR
CVF010      . LOOP ON FIELDATA DIGITS
            LSSL        A1,36       . ZERO OUT A1
            LDSC        A0,6        . MOVE NEXT DIGIT LEFT TO A1
            TNE,U       A1,05       . IS DIGIT FD SPACE?
            J           CVF020      .   YES,CVF020  NO,
            AN,U        A1,060      . REMOVE CHARACTER BIAS
            JN          A1,CVF030   . BAD DIGIT,CVF030
            TG,U        A1,10       . VALID DIGIT?
            J           CVF030      .   NO,CVF030  YES,
            MSI,U       A2,10       . DECIMAL SHIFT A2
            A           A2,A1       . ADD NEW DIGIT
            JGD         A1,CVF010   . ANOTHER DIGIT,CVF010
CVF020      . NORMAL RETURN PATH
            J           0,X11       . RETURN RESULT IN A2

CVF030      . PRINT BAD DIGIT ERROR MESSAGE, ERROR TERMINATE
            . . .
```

Here is a sample call on CVTFD:

```
            L           A0,FDNBR    . GET STRING TO CONVERT
            LMJ         X11,CVTFD   . CONVERT IT TO BINARY
```

This subroutine can be changed in several small ways to increase its utility. First, rather than diverting execution on an invalid character, it can return an error indication to its caller, which can handle the error condition as necessary:

```
CVTFD       . CONVERT FD STRING IN A0 INTO BINARY IN A2
            L,U         A1,5        . MAX CHARACTERS MINUS 1
            L,U         A2,0        . INITIALIZE ACCUMULATOR
CVF010      . LOOP ON FIELDATA DIGITS
            LSSL        A1,36       . ZERO OUT A1
            LDSC        A0,6        . MOVE NEXT DIGIT LEFT TO A1
            TNE,U       A1,05       . IS DIGIT FD SPACE?
            J           CVF020      .   YES,CVF020  NO,
            AN,U        A1,060      . REMOVE CHARACTER BIAS
            JN          A1,CVF030   . BAD DIGIT,CVF030
            TG,U        A1,10       . VALID DIGIT?
            J           CVF030      .   NO,CVF030  YES,
            MSI,U       A2,10       . DECIMAL SHIFT A2
            A           A2,A1       . ADD NEW DIGIT
            JGD         A1,CVF010   . ANOTHER DIGIT,CVF010
CVF020      . NORMAL RETURN PATH
            J           1,X11       . RETURN RESULT IN A2

CVF030      . ERROR RETURN PATH
            J           0,X11.
```

(The changed lines, all jumps, are shaded.) A normal return is now to 1,X11; the error return is to 0,X11. Here is a sample call on the revised routine:

```
            L           A0,FDNBR    . GET STRING TO CONVERT
            LMJ         X11,CVTFD   . CONVERT IT TO BINARY
            J           MAIN99      .  BAD DIGIT,MAIN99  OK,
. (GOOD RESULT IN A2)
```

The error return allows the caller to decide how to handle the error condition.

A few more changes turns CVTFD into CVTASC. But, since ASCII only fits 4 digits into a word, CVTASC will be limited to 4 digits. Note that the character bias for ASCII is the same as for Fieldata, 060.

```
CVTASC    . CONVERT ASCII STRING IN A0 INTO BINARY IN A2
          L,U       R1,3        . MAX CHARACTERS MINUS 1
          L,U       A2,0        . INITIALIZE ACCUMULATOR
CVA010    . LOOP ON ASCII DIGITS
          LSSL      A1,36       . ZERO OUT A1
          LDSC      A0,9        . MOVE NEXT DIGIT LEFT TO A1
          TNE,U     A1,040      . IS DIGIT ASCII SPACE?
          J         CVA020      .   YES,CVA020  NO,
          AN,U      A1,060      . REMOVE CHARACTER BIAS
          JN        A1,CVA030   . BAD DIGIT,CVA030
          TG,U      A1,10       . VALID DIGIT?
          J         CVA030      .   NO,CVA030  YES,
          MSI,U     A2,10       . DECIMAL SHIFT A2
          A         A2,A1       . ADD NEW DIGIT
          JGD       R1,CVA010   . ANOTHER DIGIT,CVA010
CVA020    . NORMAL RETURN PATH
          J         1,X11       . RETURN RESULT IN A2
.
CVA030    . ERROR RETURN PATH
          J         0,X11       .
```

The only differences are the JGD count put into R1, the shift count, and the test value of a space. (The labels were changed so this routine can coexist with CVTFD.)

Exercises

E21-1. Write a program to read in images of 80 ASCII characters each. (Hint: must use quarter-word mode.) For each image, edit out all spaces and numbers, and convert all upper case alphabetic characters into lower case. When the program hits an end of file (@EOF), it should print a report of the total characters inspected, the total number of characters removed, and the total number of lines.

Chapter 22: More ERs

22.1 Initial Execution Status

When the system gives control to your program, it passes in A and R registers information about the environment, such as the site ID and hardware features. The descriptions of the registers are in 3.4.4.1 of UP-4144.32, and 1.10 of UP-4144.38. Or, you can refer to this list:

- A2 and A3 contain a bit mask of hardware features, such as EIS and 1100/80 byte instructions. Please refer to your PRM for the actual bit settings.
- A4 contains the run type (actually labeled "program type"). Demand is 4, deadline batch is 5, and batch is 6. Note that the run type is not available in the PCT; it is in the Run Control Table (under the control of the Exec), and that is not accessible from a user program. The initial register contents are the only place to get this number. A program which must run in demand or batch can use this number to control the interface with the user. For example, the program might use PRINT$/READ$/TREAD$ to talk to the demand user, or COM$ to talk to the console operator in batch.
- A5 contains the current execution options in bits 0-25 (0 at the right), and a flag which is set if PRINT$ has been breakpointed.
- R1 contains the date in DATE$ format, described later in this chapter.
- R2 contains the date and time in TDATE$ format, also described later in this chapter.
- R3 contains the amount of CPU time accumulated for the run, in 200-microsecond increments.
- R11 and R12 contain the system type: '1100/60', '1100/60 EIS', '1100/70', '1100/70 EIS', '1100/80', '1100/80A', '1100/90', and so forth, in Fieldata, left justified, space filled.
- R13 and R14 contain the Exec master level in Fieldata, left justified, space filled. (Note that this is not the locally-assigned generation ID.)
- R15 contains the system site ID in Fieldata, left justified, space filled.

The option mask in A5 is the most commonly used item of those above. It is in a form called master bit notation. That's a fancy way of saying A is bit 10 (25), and Z is bit 35 (0). Since the location of each letter's bit has a strict mathematical relation to the letter's character value, you can create a mask in MASM with a simple subtraction:

```
ABIT    +       1*/('Z'-'A') . MASK FOR A OPTION
GBIT    +       1*/('Z'-'G') . MASK FOR G OPTION
ZBIT    +       1*/('Z'-'Z') . MASK FOR Z OPTION
```

With a couple of subtractions, you can see that the expressions in the example create the proper mask. ('Z'-'Z' is 0, 'Z'-'A' is 25, and so on.) Note also that it doesn't make any difference if an ASCII or FDATA character directive is in effect.

You can define a proc to create test masks:

```
P           $PROC                       .
OPMASK*     $NAME                       .
Q           $EQU        0               .
I           $REPEAT     P(1)            .
Q           $EQU        Q++1*/('Z'-P(1,I))  .
            $ENDR                       .
*           +           Q               .
            $END
```

The mask Q is initialized to zero, then successively ORed with bits for each option letter supplied on the proc call. The * on the seventh line is a "waiting label", which gives the label on the proc call the current value of the current location counter. Here is an example use of OPMASK:

```
OPTCON      OPMASK      'B','P','Q'        .
```

One last note about the Exec and your program: Before the system loads your absolute element, it zeroes out the space your program will be loaded into. This means that memory areas you reserved with $RES directives will be zero. But, it is bad practice to count on zeroes in $RESed areas. Always make sure your code initializes any data area before using it.

22.2 ER OPT$

If for some reason your program does not capture the original register contents, an ER OPT$ will retrieve the execution options:

```
            ER          OPT$            .
```

The ER requires no parameters. It returns an option mask in A0 in the same master bit notation as the initial register set, so you can use the same procs and other code to test the option settings.

22.3 ER DATE$

This ER requires no parameters, either:

```
            ER          DATE$.
```

It returns the date and time in the format shown in Figure 22-1, next page. Note that the fields are in Fieldata. Recall that since partial word modifiers are inoperative between registers, you must use shifts and ANDs to extract a field. For example, this instruction sequence retrieves the day of the month from the middle third of A0 and places it in A6:

```
            L           A5,A0       . GET FD DATE
            AND         A5,(+ 07777*/12) .AND OFF EXCESS INTO A6
            SSL         A6,12       . RIGHT JUSTIFY DAY OF MONTH
```

Or this instruction isolates the hour of the day in A1 (destroying the minutes and seconds in the process):

```
SSL         A1,24.  ISOLATE HOUR
```

22.4 ER TDATE$

This ER returns the date and time in A0, but in binary. (Contrast with DATE$, which returns time and date in Fieldata.) The fields are:

- S1 is the month (1-12).
- S2 is the day of the month (1-31).
- S3 is the year, modulo 64. In other words, a 25 in this field is 1989.
- H2 is the time in seconds past midnight.

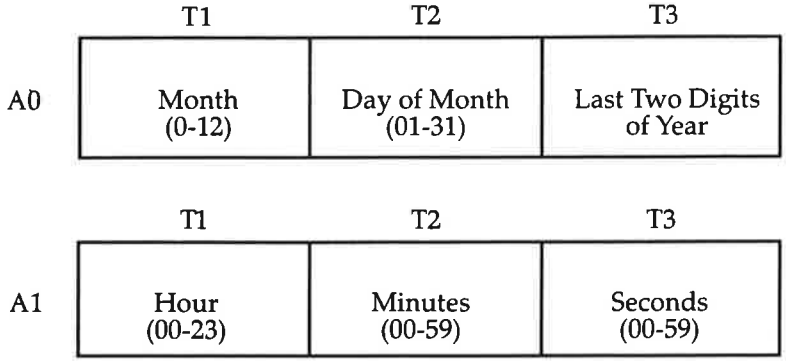

Figure 22-1. ER DATE$ Format

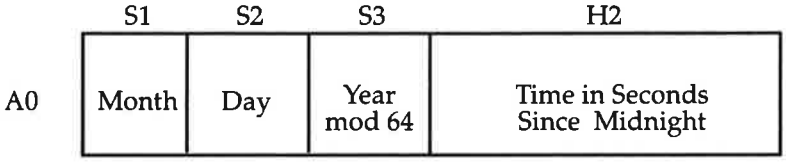

Figure 22-2. ER TDATE$ Format

While the "time in seconds past midnight" may seem like less information than DATE$ returns, it takes up less space than the Fieldata form. Use this quantity to time-stamp events, such as when a run entered a common bank, an operator made a keyin, and so forth.

Here is one way to break the time down into hours, minutes, and seconds:

```
ER          TDATE$      . GET DATE, TIME
L,U         A1,0,A0     . ISOLATE SECONDS
DSL         A1,36       . READY FOR DIVIDE (A1A2)
DI,U        A1,60*60    . GET HOURS IN A1, REM IN A2
S           A1,HOURS    . SAVE HOURS PART.
SSL         A1,36       . READY FOR MINUTES
DI,U        A1,60  . GET MINUTES IN A1, SECS IN A2
S           A1,MINS     . SAVE MINUTES
S           A2,SECS     . SAVE SECONDS
```

The second line is a simple trick to get the lower half of A0 isolated into A1. The XM portion of Xx

(here, A0) is added to the u field to make the U which goes into Aa (A1). (If this doesn't sound familiar, please review Chapter 9.)

22.5 ER TIME$

For situations in which you need a more accurate time, use TIME$. It returns in A0 the time in milliseconds since midnight. (Again, no parameters are needed.) This modification of the last code sequence breaks down the TIME$ time:

```
ER      TIME$       . GET TIME IN MILLISECONDS
DSL     A0,36       . READY FOR DIVIDE (A0A1)
DI      A0,(+ 60*60*1000) . MILLISECONDS PER HOUR
S       A0,HOURS    . SAVE HOURS PART.
SSL     A0,36       . READY FOR MINUTES
DI,U    A0,60*1000  . MINUTES IN A0, SECS IN A1
S       A0,MINS     . SAVE MINUTES
SSL     A0,36       . GET READY FOR SECONDS
DI,U    A0,1000     . GET SECONDS IN A0, MSEC IN A1
S       A0,SECS     . SAVE SECONDS
S       A1,MSEC     . SAVE MILLISECONDS
```

If you are coding common bank code, knowing the time of entry to the millisecond may be very important when debugging. Recorded in a log record or trace table, it will help you establish the exact sequence in which callers entered the common bank.

22.6 Dynamic Facilities Requests

The ERs CSF$ and ACSF$ allow your MASM program to dynamically assign a file, attach an @USE name, or perform other operations. Here is the calling sequence for CSF$:

```
L       A0,(<words>,<buffer>)  .
ER      CSF$                   .
```

The parameters in A0 are, in reverse order:

- The address of a buffer containing a control statement in Fieldata. The list of allowed control statements is below.
- The length of the image in words. The maximum is 40 words; if the value given is 0, the system will assume the image is 14 words.

Most of the statements you can pass on ER CSF$ are facilities requests. These are:

```
@ASG
@USE
@CAT
@FREE
@MODE
@QUAL
```

There also miscellaneous statements accepted via CSF$:

```
@ADD
@BRKPT
@LOG
@SYM
@START
@CKPT
@RSTRT
```

(Before you get the idea that the list is arbitrary—note that there are no FURPUR statements, for example—what these statements have in common is that they do not cause a program load. They are processed directly by the Exec.)

What does a typical CSF$ call look like? Here is an example:

```
ASGIMG    '@ASG,A  TEST*WORK.'   .
ASGLEN    $EQU       $-ASGIMG    .
.
          L          A0,(ASGLEN,ASGIMG)  .
          ER         CSF$        .
```

That is the "book" way, using both parameters in A0. You can take advantage of the way the Exec scans images by putting a space period space (" . ") terminator at the end of the image. This technique allows you to ignore the word count parameter, as long as your image is no more than 14 words long. Here is the above example, changed to use the image terminator:

```
ASGIMG    '@ASG,A  TEST*WORK.  .  '  .
.
          L,U        A0,ASGIMG   .
          ER         CSF$        .
```

The L,U instruction is easier to code than the line containing the literal in the first version. Just don't forget the space period space terminator in the buffer, or the Exec may reject an otherwise valid image.

Code a call on the ASCII equivalent of CSF$, ACSF$, exactly the same:

```
          L          A0,(<words>,<buffer>)  .
          ER         ACSF$       .
```

The only difference is that the image must be in ASCII. The maximum length is 60 words; if <words> is 0, 21 is assumed. The same image terminator technique applies, as well.

22.6.1 Checking CSF$/ACSF$ Return Status

Both ERs return a status in A0. In the case of facilities requests (@ASGs, @USEs, and such), the status is a bit mask. Each bit can signal a portion of the problems the Exec had with satisfying your request. The leftmost bit, 0 (35), is set to signal that the request was rejected. You can use a simple instruction to transfer control on a rejected request, as in the following example:

```
L,U         A0,ASGFL1   . ADDRESS OF IMAGE BUFFER
ER          ACSF$       .
JN          A0,PH2099   . REJECTED,PH2099   OK,
```

Since bit 0 (35) is the sign bit, a Jump Negative will jump when the Exec has rejected your request. (In the code that acts on the rejection, be sure to save the status before printing an error message—wiping out A0.)

You can check the meanings of the other bits in the status in Appendix C of your PRM (UP-4144.32), Appendix B of UP-4144.38, or section 3.2 of your Hardware/Software Summary, UP-7824.3.

To check for bits other than 0 in the status, use a sequence like this:

```
AND         A0,(1*/19)  . FILE ROLLED OUT?
JNZ         A1,BGN070   .  YES,BGN070  NO
```

This sequence checks bit 16 (19), which signals that the requested file has been rolled out.

The other, non-facilities images accepted by CSF$ and ACSF$ pass a different set of statuses. They are in nearby sections of the same reference manuals. Note that if an @ADD image is not accepted, the Exec does not pass back a status, but creates a contingency condition, usually error terminating your run.

Exercises

E22-1. Write a program which will display the following information from the initial register contents:

Date
Time
Accumulated CPU time
System site ID

Chapter 23: Collecting Your Program or Before The Program Executes

23.1 Check Your Assembly

A number of things can go wrong with your assembly, including:

- There may be assembly error flags. MASM will list in its summary line the number and types of error flags it generated. Locating all the images in the print file that begin "END" will print summary lines from most system processors, including MASM and MAP.
- The relocatable output file may have overflowed, preventing MASM from finishing the assembly. This error will not show up in MASM's summary line.

If the collector prints a message to the effect that "the relocatable element contains errors", go back and look at the assembly listing, even if the summary line looks clean.

23.2 Get To Know The Collector

Chapter 5 covered some of the basic ECL you need to get your program assembled, collected, and executed. This chapter will build on Chapter 5, and go into more detail about the collector, or @MAP processor. Expertise you develop with the collector will apply as well to Cobol or Fortran as to MASM. In fact, as you grow in experience with MASM, you will appreciate, more than ever, the job done by any high level language system.

23.2.1 Overview of the Collection Process

Previous chapters have explained what location counters are and how they are used. At collection time, we think of them not as the actual counters used at assembly time, but contiguous chunks of memory. Each element in a collection consists of one or more location counters, some odd, and some even. The collector, by default, places odd location counters (1, 3, 5, etc.) in the I bank, and even location counters (0, 2, 4, etc.) in the D bank. The reason for this convention goes back to the hardware design of the 1108. The 1108 was able to overlap instruction and data fetches when the instructions and data were physically separated into different memory cabinets.

Collection is the process of stringing location counters, one after the other, to make an absolute element. Each time an external reference is encountered, the collector adds it to a list for a library search. The collector searches the files specified for library searches in the order they are given. If the collector can not define an external reference and it has run out of files to search, it assigns that label a value of zero and produces an error message.

The collector tends to accumulate library elements in a "pushdown" fashion. In other words, it adds library elements to the list of elements to be collected, but ahead of the user-chosen elements (that is, at lower addresses). Each additional library element is placed at the top of the stack, displacing earlier library elements and user elements to higher and higher addresses. The reason for this scheme is simple: if a subscript or index in a user program gets out of bounds, it will not wipe out information in a library element.

Having built a list of the elements to collect, the collector then begins to assign addresses. It accumulates a program-relative address for the beginning of each location counter, adding the length of each new one to the accumulated total.

If you have ever done an L-option assembly, you will notice that beneath each line on the left side, is a note, something like "17-0 + LC 00". What this means is that the collector must add to the contents of bits 17-0 (H2) in this word, the starting address of location counter 0 of this element. The generated code shows the address of the operand relative to the start of the location counter; adding the beginning address of the location counter at collection time completes a program-relative address. The collector, as it retrieves each relocatable element, does that addition for each field that needs it. The addition of the start of a location counter is called relocation.

23.2.2 Basic Collector Directives

There are five directives that will get most programs collected. You will find them in your Quick Reference Guide, UP-7824.3, on page 1-63.

- IN requests inclusion of a list of elements in an absolute.
- FRSTIN does the same job as IN, except it forces the named element to the beginning of each bank. Only one element may be FRSTINed; the collector will ignore any others.
- NOT disallows the collector from collecting a named element list or elements from a named file.
- LIB tells the collector to do a library search.
- END tells the collector there are no more directives in the source. It does not need any parameters.

Collector directives are free-form, but it improves their readability if you use the same tab settings as for MASM. That is, put any of the directives above in column 11, and the next field (usually a file or element name) in column 21. You can even put comments, preceded by a space period space terminator (" . "), in column 39.

23.2.2.1 INcluding Files and Elements

The IN directive tells the collector to retrieve one or more elements for inclusion in the resulting absolute element. Here is the format:

```
IN          name,name,...
```

Each name may be an element, or, if the name ends with a period (that is, a file name), the collector will collect every relocatable in the file. The author has found that naming more than one element on an IN directive can be confusing; it also makes changing the order of those elements difficult.

23.2.2.2 FRSTINing An Element

Forcing a single element to the beginning of the collection can be useful when you are debugging a MASM program. Nearly every MASM program the author creates uses FRSTIN. FRSTIN has the effect of forcing $(0) of the named element to start at 040000 and $(1) to 01000. If there is a critical table, buffer, or packet, it is easier to find a field when the addresses start at a nice round number. More importantly, as you debug the program, it may be necessary to add and remove various library routines from the collection. Having the main program element at the same address every time makes it easier to go to the right address in a snap or post mortem dump. The collector honors a FRSTIN for only one element.

23.2.2.3 NOTing Out A File Or Element

By default, the collector pulls in relocatables from TPF$. If you use the kind of ECL shown in Chapter 4, and you work on several utilities in a session, you may end up with the relocatables of more than one in TPF$. If the collector attempts to collect more than one main program, it will give you error messages like "start address not used". You can use a directive like:

```
NOT         TPF$.
```

to keep unwanted relocatables out of your collection. The NOT will not override an IN TPF$.name, so you can still include as much as necessary from TPF$.

23.2.2.4 LIBrary Searches

The collector is already set up to search a number of files for external references. Most of the files have qualifiers like SYS$ or SYSLIB; they contain Cobol, Fortran, Plus, or other language subroutines. If you need to include subroutines outside of your own program file, you can name the files they are in by means of a LIB directive:

```
LIB         file,file,...
```

The files you name on a LIB directive must have an entry point table. That is, you must use the FURPUR @PREP statement on the file. The entry point table saves the collector from the overhead of having to search every relocatable in a file to locate an entry point definition. The PREP statement is simple:

```
@PREP   file.,file.,...
```

Each time a new relocatable is added to a file, the entry point table (if any) is cleared. Don't forget to PREP the file again before it is used for a library search. You don't need a PREP to INclude elements from a file, though.

Chapter 24: Executing Your Program

24.1 How to Execute Your Program

There are two ways to get the Exec to execute your program:

```
@XQT,options     file.elt
```

or:

```
@file.elt,options     spec,spec,...
```

The first is the normal mode for applications programs. The file name may be expanded to the full-blown OS 1100 file nomenclature:

```
@XQT,options    qual*file(fcyc)/read/write.elt/version(ecyc)
```

or you can use anything in between. The operating system does not attach any meaning to the options on an @XQT statement; that is up to the program.

The other form is a bit more interesting. When a program is called as a processor:

- @file.element, or just
- @element,

you can feed specification fields to it. Each of those spec fields is subject to the limitations of the full-blown element specification as shown above. But the information in a spec field is not required to be an element or file name. It can be a block count, a disk unit name, a device type, or anything which satisfies the Exec's syntax checking.

The processor call format is included here because it lets you pass your program input data without the usual PRINT$/READ$ rigmarole. To be sure, if your program does a READ$ when there are spec fields, the specification information will be passed to your program. It is in a format called INFOR$, INternal FORmat. You can look up INFOR$ in your SYSLIB manual. There are several system-defined subroutines which let your program interpret and search the INFOR$ information.

The points here are:

- Processors are not magic; they just include code to read the INFOR$ table.
- If your program is written to read control information from the runstream, and is normally executed by @XQT, don't call it with @PROG. The program, on the first few reads, will get INFOR$ data, which is just gibberish without the proper interpretation.

Chapter 25: Program Error Analysis or Picking Up The Pieces

25.1 Types of Errors

Once you have a clean assembly (no error flags, no undefined symbols in the collection), you can ask the system to execute your program. (The system will not enforce it, but the clean assembly should precede the execution.) If it does not work, it may be for one of many reasons. Failures you will encounter fall into one of three classes:

- Program contingencies. The operating system has caught your program doing something it cannot allow.
- Errors detected by a system library element. System library-detected conditions are as close as a MASM program gets to the kinds of errors detected by a language system's runtime library. The system library assumes a much smaller role in detecting error conditions in MASM programs.
- Functional or logic errors; your program simply did not do what it was supposed to, but did not "blow up".

You can probably tell which type of error your program got:

- If your program hit a contingency, your print file was decorated with a register dump and related data.
- Most of the system library routines have error returns of some sort, and your program can respond to them in some way.
- Finally, you know if the program did not "give the right answers" or did not take the proper actions, such as reading a file, displaying totals or whatever.

The remainder of the chapter will be concerned with the first and third types of errors. (The system library routines are beyond the scope of this book.)

25.1.1 Program Contingencies

Webster's dictionary defines a contingency as (among other things) "something whose occurrence depends on chance or uncertain conditions". In the OS/1100 environment, a contingency is an error condition trapped by the operating system or the hardware. You've requested the machine or operating system to do something it can't do without interfering with other users or violating other rules. Sometimes, the hardware, in its normal process of validating operands of instructions, etc., alerts the Exec with an interrupt. The Exec has to identify the interrupt type (e.g., type of offense), the active user, and the action to be taken for the user. Other times, it's a portion of the Exec that finds something wrong, but similar action has to be taken.

There are 16 contingency types, and maybe a couple of hundred subtypes within one of those major

types (error mode). The most common contingencies are illegal operation, guard mode, and error mode.

An illegal operation contingency (contingency type 1) occurs when the machine tries to execute data, such as a binary zero, as an instruction. For safety, in fact, the commonly occurring values positive zero (00) and negative zero (octal 77) are and probably will be forever excluded from the 1100 instruction set. What causes an illegal operation to occur? If a bad index is used to access a character load vector, execution can stray into data; or, if an LIJ or LMJ return address is corrupted, then used for a return, who knows what the machine will find upon return?

A guard mode contingency (contingency type 2) is also detected by the hardware. Series 1100 CPUs have two basic modes: guard mode and Exec mode. In guard mode, the hardware enforces limits on what programs can look at or do. A sort of "fence" (addressing limits) are drawn around the program, preventing it from accessing memory areas outside the limits; the program can only "see" the user register set; and, the program can not execute a number of I/O and other instructions. (On 1100/90 systems, there are a number of user and operating system machine states or privilege levels, but for our purposes, they boil down to guard mode and Exec mode.)

A guard mode contingency occurs when a user program tries to violate one of the hardware rules: no privileged instructions, no access outside the storage limits, no write access to read-only banks, and so forth. Easily the most common cause of a guard mode contingency is a subscript or index register going out of range. A close second is an attempt to reference an undefined variable (read/write access) or an undefined subroutine (jump-to access). Either one of these would cause a reference to a zero u field. Since there are multiple types of guard mode contingencies, depending on the exact nature of the violation, the Exec will tell you the error code, as well, when the problem occurs.

The Exec actually presents three indicators when it detects a contingency: the contingency type, the error type, and the error code. Think of these three values as a hierarchy, with the first at the top. Within contingency type 2 (guard mode) is an error type 0, with almost a dozen error codes within it. (These are listed in the Exec PRM, UP-4144.32, in section 4.9.2.2; UP-4144.38, in section 25.2). The error code will tell you the exact offense committed by your program: storage limits/write protect violation, control register violation, etc.

Contingency type 012, called error mode or Emode, is the third common contingency type. The Exec gives your program an Emode contingency when it can't satisfy a request, and it is clear your program could not go on without the request being satisfied. There are ten or so error types within contingency type 012. The error types are divided into functional areas of the Exec — I/O, symbiont, TIP, etc. Each error type defines up to 63 error codes. For I/O, some of these codes double as normal completion statuses. All of the Emode error codes are defined in the PRM (UP-4144.32, Appendix C; UP-4144.38, Appendix B).

The hierarchical nature of contingency type, error type, and error code cannot be emphasized enough — you want to be sure to look up the right error number when your program fails. Some MASM programs register for contingency handling. When the Exec passes control to the contingency-handling code, it puts the contingency type, the error type, and error code in a packet, normally dumped by the program's contingency handler. But the contingency type is actually the third six-bit field of the three, not the first. Don't let that throw you.

25.2 Confronting the Error Termination

When an error termination occurs, the Exec automatically displays a number of things that show what the program was up to when it died. Let's take these items one or two at a time:

- The contingency type, error type and error code: these tell what went wrong from the Exec's perspective.
- The error address and BDI: these tell where the problem occurred. You can refer to your program's collection listing or a PMD to find out in which element the problem occurred.
- A dump of the D (or designator) bits associated with the offending activity: The only one of these you might be concerned with is for quarter-word mode. If your program doesn't toggle quarter-word mode, don't worry about the D bits.
- A list of the currently based banks: these tell which and what kind of banks were based on the four BDRs. For now, your programs will have only one I bank and one D bank.
- The user jump history stack: this supplies varying amounts of information about the jumps, LMJs, and LIJs done by the program. If your program did a number of jumps, and there were enough LMJs done to leave "tracks" behind for you to follow, you may be able to use the jump history to make sense of what happened. Or, if your program did a straight jump into a data area with no other trail to follow, the jump history can give information obtainable no other way. On the other hand, the jump history is not maintained in the Activity Save Area (ASA) as are register contents, and the size of the jump history stack varies by CPU type. The author does not count on much help from the jump history.
- A dump of the user registers: use this to see the data the program was working on when it died, and possibly some evidence of where the program had been (that is, "tracks").

25.2.1 Problem Solving

Once you've had the error termination, there are a few things you can do before re-running for a PMD (or running for a systems type to help out). A close inspection of the error dump (just described) could get you back on track.

Take a look at the error address. Looking at your collection listing, find the element which includes the error address. Look for the element whose odd location counters occupy a range encompassing the error address. Note which location counter is involved.

Once you have identified an element, the next step is to figure out where in the element the error occurred. There are two ways to figure out the element-relative address of the error:

- If you can do octal arithmetic, or have access to an octal calculator, subtract the beginning address for the involved location counter from the error address. The difference gives the offset from the start of the location counter of the element to the error address.
- Without octal arithmetic, run again, if necessary, to get a PMD, locate the error address in the dump, and note the offset from the beginning of the location counter.

Using the register dump, look to see what was in the registers your program was using at the failure point. The leftmost column of registers, A, X, and R, represent the 0-numbered registers, the next column 1-numbered, and so on. (X0 and R0 have special purposes. Even if they are printed, you should ignore them for now.) The first row of each group is numbers 0 through 7, the second row 8 through 15. (The A registers include 4 additional registers in a third row. They are numbered A15+1 through A15+4. You can safely ignore them for now.)

Pay particular attention to the contents of the index registers used by your code. If one of them is pointing somewhere other than meaningful data, your program may be experiencing a classic case of GIGO: Garbage In, Garbage Out. If inspection of the registers themselves does not point the way, you may need to look at your program's data areas.

If the reentry address happens to fall inside a library element, chances are good it is still your code's fault. Look at the contents of X11, which is the LMJ register used by most library routines. It probably points to the location in your element where the routine was called. Even if the registers your code was using before the call have changed, you can still double check all the parameters you passed to the library routine. A failure in a library routine almost invariably points back to a weakness in the calling element.

25.3 The Post Mortem Dump (PMD)

Each time a task (user program or processor) terminates, the Exec can save a copy of the program for later inspection. The memory image of the program and other information about its origin go to a file called DIAG$, automatically assigned for you when your run starts. Writing the program to DIAG$ is not done for every program. DIAG$ writes are normally suppressed for processors; run statement options can affect DIAG$ writes. You can suppress all DIAG$ writes with the @RUN,/N option, or cause DIAG$ writes for all programs, user and system, with the @RUN,/Y option. (Avoid the Y option unless debugging a system processor.)

Still another @RUN option, W, can request a write to DIAG$ and an automatic PMD whenever a program terminates in error. Note that the W option does not give you any control over the options given to PMD.

The PMD processor itself is fairly easy to use; although it can accept fairly complicated specifications, you often don't need anything fancy.

- @PMD with no options requests a dump regardless of whether the program terminated in error or not; it dumps non-system-library elements within the absolute, both odd (I bank) and even (D bank) location counters.

- The E option requests that the PMD be printed only if the program terminated in error.

- The L option requests printing of all elements, including system library elements. (Normally, you don't need to dump them.)

- The P option requests a dump of the PCT at termination, in addition to anything else dumped. You probably won't need a PCT dump unless you need to know which and how many files were assigned to your run, and you know how to follow the PCT's file chain.

PMD can be called multiple times after a single program execution, and it can accept specifications

of bank names, element names, and location counters. It can also dump areas in Fieldata or ASCII, and accept a Fortran format image to control the printing. (The latter would be a boon to the programmer using floating point variables.) The hint here is that you could call for a general PMD to get everything down in octal, then dump one or two location counters in ASCII to zero in on working storage or whatever. PMD can also print the dump in a variety of other formats.

So, assume that you have put an @PMD,E in your ECL as the first control image after the @XQT of your program. (There may be intervening data.) Re-run the ECL if necessary. The Exec will execute your program, which terminates in error, and PMD produces a dump which appears just after the register dump in the print file. Let's examine the dump in detail. (See Figure 25.1, page 25 - 9 for a sample PMD.)

For each location counter (contiguous memory area) dumped, PMD prints an identifying line. Listed in this line are the bank name, the segment name, the element name, the location counter number, and the start location of the location counter. The last field of the line is easy to miss — the date and time of creation of the element. If for some reason, your system or local subroutine library elements are out of date or out of sync, the date and time are crucial.

Following the identifier line, each line has 10 columns. The first column contains the location counter value itself, that is, a number that starts at 00000 and is incremented 010 for each row. The second column contains the program-relative address, and it, too, is incremented 010 for each row. The remaining 8 columns show the octal contents of 8 words of memory.

PMD repeats this format for each of the location counters it dumps. PMD may dump one additional area, not allocated under a location counter. There are two ways this area may exist:

- If the program did an ER MCORE$ to request additional memory, PMD will dump this area.
- There may be a difference between the last address allocated by the collector for a bank and the granularity of the Exec's allocation, which may be 8, 64, or 512 words. (You have some control over this granularity at collection time via a collector directive.)

Either way, PMD labels this area "dump beyond allocated limit"; many programs, including language libraries, use MCORE$ and put dynamically allocated tables and buffers there. It is most often allocated at the end of the D bank.

25.3.1 Interpreting Instruction and Data Words

You may have occasion to take an octal instruction word, and turn it back into a readable instruction mnemonic. Such a process is called disassembly, at least when applied to a whole element. It is unlikely you will have to disassemble more than a few instructions, though.

To interpret an octal instruction word, write down the digits, separated by about 3 spaces between each pair, as in this example:

```
1   0   7   3   6   0   0   4   0   0   7   2
```

Remember the instruction word format: f, the major function or "op" code, is 6 bits; j, a, and x are 4 bits each; h and i are 1 bit each, and u is 16 bits. Since f is 6 bits, you can see it directly. Next, write

down the binary equivalent of next 4 octal digits, 7360, underneath the corresponding octal digits:

```
1    0    7    3    6    0    0    4    0    0    7    2

          111  011  110  000
```

Next, regroup the twelve bits into 4-bit "nibbles" (j, a, and x), and write their octal equivalents beneath:

```
          1110 1111 0000

            16   17   00
```

Finally, look up the f value of 10. It is LA, Load Accumulator. A j value of 016 is U, and an A value of 017, given the op code, is A15. Now you can write down the entire instruction:

```
          L,U          A15,040072  .
```

Look at the U field of 040072. Recall that since your D bank starts by default at 040000, the instruction refers to data at 072 relative to the lowest location counter in the D bank. If that is in your element, look at that location to find the label whose address is being loaded.

Data words are much easier. Just remember that the partial words fall on even octal digit boundaries:

- Half words are 6 octal digits long.
- Third words are 4 octal digits long.
- Quarter words are 3 octal digits long.
- Sixth words are 2 octal digits long.

25.4 Other Debugging Aids

Chapter 12 explained how to use the memory snapshot ER, SNAP$, and its companion proc, L$SNAP. Use snaps generously on critical tables, but be sparing in dumping large buffers. When you code a snap, be sure that you know exactly how you will use the data it displays. A snap must be able answer important questions to be justified.

If the data which is most important is buried in the middle of a large buffer or table, don't snap the whole thing. Use a little simple logic to locate the important part, and set up the snap packet to dump just that. Chapter 12 showed how to change the snap address for an L$SNAP. If you no longer need the code, but think you may need it again, proc it off with an $IF DEBUG/$ENDF pair (but don't forget that if DEBUG is undefined, MASM will consider it zero).

A snap in the middle of a loop may generate too much paper to be useful. Or, it may be tricky to locate the important data in the middle of a snap. Code up a simple sequence to print a trace line each time you pass an important point in the code. That can make it easier to debug your program from a demand terminal, too. You may not even need to print the breakpoint file, just scan it with an editor in read-only mode.

If the error occurs after the program has been running for hours, and it is not practical to use snaps, there is another way. Create a trace table in your program. Each time an important event occurs, record the pertinent fields in a new entry. If there is no limit on the number of times the event can occur, make the table circular—that way, you are assured of getting the last "n" occurrences of the event, and you will never run out of memory. Chapter 20 showed how to create a circular linked list. (Hint: make the length of each node a multiple of 8, so each field will appear in the same column of a PMD.)

Still another way to stop errors is by using traps. If a certain X register must not be more than 200 at a certain point (or you suspect it might be), code a simple test like this:

```
S           X5,A15      . SET UP FOR TRAP.
TG,U        A15,200+1   . X5 > 200?
ER          EABT$       .   YES,STOP_COLD
```

This technique is strictly for short-term use! While you are debugging, you are much more aware of what can go wrong with the program. An error abort will not take you by surprise. Once the program has been working for a few months, an ER EABT$ out of the blue may be as much of a surprise to you as to the people using your program. If the condition is a "valid" error condition, make the error termination a bit more formal. Print an error message in English, and snap the offending registers, tables, or whatever.

25.5 FLIT

The FLIT processor (Fault Location through Interpretive Testing) is another tool for checking out your program. Space limitations prevent a full discussion of FLIT, but here are the high points:

- FLIT can simulate any 1100 processor.
- FLIT can load your program from an absolute element, and execute it under your control, tracing instructions and register contents as it goes. FLIT commands allow you to inspect memory locations while the program is executing, as well.
- FLIT can also load your program from DIAG$ if the original absolute element is still available. You can write FLIT functions to dump out and/or test every memory location in your program, if you need to. In other words, if you have a complicated data structure that takes 20 minutes to interpret by hand from a PMD, you can use FLIT to break it down and format it automatically after every execution.

You can read more about FLIT in documents listed in Chapter 1.

25.6 Recapping The Strategy

The first time your program hits an error termination, you may not have set up for the PMD to be printed. It's a good idea to use what is available that first time to extract whatever can be used immediately — the error address, register contents, etc. If resolution is urgent, and turnaround slow, pull out all the stops: put snaps on every critical table, and put an @PMD,LE in the runstream to get a dump that will be sufficient 90% of the time. Don't bother with a PCT dump unless there is a clear need for it.

Once you've got all these pieces in hand, the story is almost certainly in front of you. Don't lose your nerve, and be prepared to change something in the program or collection to get a different look at the situation. You can do it!

Exercises

E25-1. Locate copies of the documents for PMD and FLIT for the versions installed at your site. In the PMD manual, look over the processor options and ways to use Fortran-style formats. Note the way to dump a single location counter. In the FLIT manual, look at the command(s) for loading programs, both from an absolute and from DIAG$.

```
@PMD

PMD 32R2 S74.33 03/23/89 14:54:30

 DUMP AFTER EXECUTION OF PROGRAM: BITMOVE
   CREATED ON: 23 MAR 89 AT 14:54:28
   COLLECTED WITH SYS$*RLIB$ LEVEL: NONE

BANK- $IBANK          SEGMENT- $MAIN$          ELEMENT- BITMOVE          $( 01)  AT ADDRESS  001000   CREATED ON: 23 MAR 89 AT 14:54:27
000000  001000    107000040004  100020040011  467320004400  374000000014  010000001012  107000001010  724400000126  742000001013
000010  001010    071631222433  200017040000  000000000000  724400000011

BANK- $IBANK        DUMP BEYOND ALLOCATED LIMIT
000000  001014    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
   LINES OF 000000000000     SKIPPED     000750 OCT WORDS      000488 DEC WORDS
000760  001774    000000000000  000000000000  000000000000  000000000000

BANK- $DBANK          SEGMENT- $MAIN$          ELEMENT- BITMOVE          $( 00)  AT ADDRESS  040000   CREATED ON: 23 MAR 89 AT 14:54:27
000000  040000    124110111123  040111123040  101040124105  123124040040  000000000000  000000000000  000000000000  000000000000
000010  040010    000000000000  000000440000

BANK- $DBANK        DUMP BEYOND ALLOCATED LIMIT
000000  040012    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
   LINES OF 000000000000     SKIPPED     000750 OCT WORDS      000488 DEC WORDS
000760  040772    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
```

Figure 25 - 1. Sample PMD

```
@MASM,L   MDM.BITMOVE,BITMOVE
MASM 4R1A (840907 0128:18) 1989 Mar 23 Thu 1454:27
                                        100.       $(0)      AXR$       .
                                        110.                 ASCII                .
    0 000000  124110111123              120.       FROMBUF   'THIS IS A TEST'     . SOURCE BUFFER.
      000001  040111123040
      000002  101040124105
      000003  123124040040
              000000000220              130.       FROMLEN   EQU        ($-FROMBUF)*36   . BIT COUNT.
      000004                            140.       TOBUFF    RES        5                . DESTINATION BUFFER.
                                        150.       $(1)      .
                                        160.       START     .
    1 000000  10 16 00 00 0 000004      170.                 L,U        A0,TOBUFF        . SETUP 'TO' REGISTER.
           **      BITS 17-0 + LC 0
      000001  10 00 01 00 0 000011      180.                 L          A1,(FROMBUF+1*/17) . SETUP 'FROM' REG.
           **      BITS 15-0 + LC 0
      000002  46 16 15 00 0 004400      190.                 LXI,U      A1,FROMLEN*/4    . BIT COUNT FIELD.
      000003  37 10 00 00 0 000014      200.                 BIM        A0               . DO THE MOVE.
      000004  01 00 00 00 0 000012      210.                 L$SNAP     'BITMOV',2,15,FROMBUF . SNAP RESULTS
           **      BITS 15-0 + LC 1
      000005  10 16 00 00 0 000010
           **      BITS 17-0 + LC 1
U     000006  72 11 00 00 0 000000
           **      BITS 15-0 + XR SNAP$
      000007  74 04 00 00 0 000013
           **      BITS 15-0 + LC 1
      000010  071631222433
      000011  2 00017 000000
           **      BITS 17-0 + LC 0
      000012  000000000000
U     000013  72 11 00 00 0 000000      220.                 ER         EXIT$            . QUIT.
           **      BITS 15-0 + XR EXIT$
              000000                    230.                 END        START            .
           **      BITS 15-0 + LC 1
    0 000011  000000400000
           **      BITS 35-0 + LC 0

LOCATION COUNTERS:    $(0)  000012    $(1)  000014

EXTERNAL REFERENCES: EXIT$          SNAP$

CROSS REFERENCE LISTING *=LABEL, **=LABEL*, D=DIRECTIVE
AXR$              100D
EXIT$             220
FROMBUF           120*      130       180       210
FROMLEN           130*      190
L$SNAP            210D
START             160*      230
TOBUFF            140*      170

END MASM  -  LINES: 195  TIME: 3.228  STORAGE: 5005/1/1/8013
```

Figure 5 - 1. Sample MASM Listing

About the Author

Mike Maddox joined the Univac Division of Sperry Rand in July of 1973 in an assignment at Bell Laboratories. Since then, he has had assignments at some of Sperry and Unisys's largest customers, including the Internal Revenue Service, the U. S. Department of Agriculture, and the Texas Department of Public Welfare (now Department of Human Services). He has done a variety of programming, ranging from systems software development (on BICS-1100 at Bell Laboratories) and systems software support at several sites, to transaction program development at other sites.

Mr. Maddox uses many different languages in his work, including his favorite, SSG, as well as MASM, Cobol, Fortran, Turbo Pascal, and embedded languages in other products. A PC and WordPerfect enthusiast, he is active in a PC user's group, and currently plans to take aim at the language C.

His writing experience began in 1985 with a magazine that covers the Unisys market, and presently has a monthly column, "The 1100 Domain", in Unisys World.

About the Book

Original entry of the text was on an Everex 1800A 80286-based PC, using Word Perfect 5.0. ASCII and generalized word processing-format files were imported into Pagemaker by the publisher. The original drawings were produced using Gem Draw 3.0 and re-drawn by the publisher's staff.

The body typeface is 11 pt. Times, and the chapter titles are set in 24 pt. Times bold.

Index

$ANDF 18-4, 18-5
$ASCII 4-9, 4-10, 12-4, 13-10, 16-12,21-7
$DO 4-3, 4-7, 18-1, 18-2, 18-5
$ELSE 18-3, 18-4
$ELSF 18-4, 18-5
$ENDF 16-11, 18-3, 18-4, 25-6
$ENDR 4-11, 16-10, 16-11, 18-2, 20-4, 22-2
$EQU ..... 4-3, 4-7, 4-8, 4-9, 4-10, 9-4, 9-5, 9-7, 9-10, 10-2, 13-11, 13-13, 15-1, 15-2, 15-3, ... 15-5, 15-6, 16-7, 17-1, 18-2, 19-2, 19-3, 20-4, 20-5,22-2
$EQUF7-6, 7-7, 9-4, 9-5, 9-6, 9-7, 9-12, 10-2, 13-7, 15-3, 15-5, 15-6, 16-8, 19-2, 19-3, 20-4, 20-5
$FDATA 4-7, 4-9, 4-10, 11-10, 12-4, 13-7, 13-11, 13-13, 16-12
$FORM 4-3, 4-6, 4-7, 9-10, 9-11, 9-12, 12-5, 18-1
$GFORM 16-12, 18-1
$GO 16-11
$IF 16-11, 18-3, 18-4, 18-5, 25-6
$INFO 21-1
$LIST 16-10, 16-11, 20-4
$LIT 9-10
$NAME 16-3, 16-4, 16-5, 16-9, 16-10, 16-11, 22-2
$OFF 18-3
$ON 16-11, 18-3, 18-4
$PROC 16-1, 16-2, 16-3, 20-4, 21-3, 21-4, 21-5, 22-2
$REPEAT 4-3, 4-7, 4-11, 16-10, 16-11, 18-2, 18-6, 20-4, 22-2
$RES 4-8, 6-11, 15-1, 15-2, 15-3, 15-5, 15-6, 17-1, 19-2, 19-9, 20-4, 22-2
$UNLIST 16-11, 20-4
@ASG 5-4, 13-8, 13-9, 22-4
@BRKPT 5-4, 5-5, 13-3, 13-8, 13-9, 22-5
@PDP 2-6, 19-3
@PREP 23-3
@XQT 5-2, 5-3, 13-3, 17-2, 24-1, 25-5
1100/60 ..... 2-2, 3-2, 7-4, 21-6, 22-1
1100/40 ..... 2-2, 3-2
1100/90 ..... iv, 2-2, 2-3, 3-2, 3-7, 9-10, 21-6, 22-1, 25-2
1100/80 ..... 1-1, 2-2, 3-2, 21-5, 22-1
1108 ..... 2-1, 2-2, 3-2, 4-4, 23-1
1110 ..... iii, iv, 2-1, 2-2, 2-3, 3-3, 4-4, 21-6
2200 ..... iii, iv, 2-3, 3-2, 3-3, 3-6, 6-1, 19-7, 21-1, 21-6
A$PRINT 13-6, 13-10, 16-2, 16-3
A$PRNTA 13-9, 13-10
A$READ 13-5, 13-10
A$READA 13-9, 13-10
AA 6-4, 6-5, 7-6
ABORT$ 12-2, 16-13
ACSF$ 16-6, 22-4, 22-5, 22-6
algebraic shifts 11-1, 11-2
ANA 6-5, 7-6
AND 3-2, 11-3, 11-4
ANU 6-6
ANX 6-5

APRINT$ 13-6, 13-10, 13-12, 16-2, 16-3
APRNTA$ 13-9, 13-10
APRTCN$ 13-12
AREAD$ 13-5, 13-10, 13-14
arithmetic operators 17-1, 17-2
ASCII ..... 2-4, 3-6, 3-7, 3-9, 4-9, 4-10, 4-11, 9-11, 12-4, 13-1, 13-4, 13-5, 13-6, 13-9, 13-10, ... 13-13, 14-7, 16-12, 17-4, 19-5, 20-11, 21-4, 21-5, 21-6, 21-7, 21-8, 21-9, 22-1, 22-5, 25-5
AU 6-6, 19-8, 19-9
avail list 20-3, 20-4, 20-10
AX 6-5
AXR$ 4-2, 4-4, 4-5, 19-9, 21-6
basic ECL 5-1, 23-1
BIM 21-6
bit-width fields 11-8
BT 15-1, 15-2
CASEOF 16-12, 16-13
CAU 2-1
changing location counters 16-8
character ..... 2-2, 2-4, 3-7, 3-8, 3-9, 4-1, 4-2, 4-7, 4-9, 4-10, 4-11, 9-10,11-7, 12-2, 12-4, 13-1, ... 13-4, 13-5, 13-6, 13-10, 13-12, 16-1, 17-4, 18-1, 19-1, 19-6, 19-7, 21-1, 21-2, 21-5, 21-6, ... 21-7, 21-8, 21-9
character vectors 21-2, 21-3, 21-4, 21-5
circular shifts 11-1, 11-3
circular list 20-2
collection 1-3, 4-4, 4-9, 12-1, 16-12, 21-1, 23-1, 23-3, 25-3, 25-8
collector 2-6, 4-4, 4-9, 5-2, 5-3, 5-4, 7-4, 12-1, 14-1, 23-1, 23-2, 23-3
comments 4-1, 4-2, 8-2, 9-2, 11-3, 14-4, 15-1, 16-13, 19-2, 19-5, 19-6, 19-7, 23-2
conditional expressions 17-6
contingencies 25-1, 25-2
contingency 15-5,18-9, 22-6, 25-1, 25-2
CPU ..... iii, 2-1, 2-3, 2-4, 3-1, 3-2, 3-3, 3-4, 3-9, 3-10, 6-1, 6-2, 6-3, 6-4, 6-6, 6-10, 7-1, 7-2, ... 7-3, 7-6, 8-5, 8-9, 8-11, 9-1, 9-6, 9-7, 9-8, 11-1, 14-5, 15-1, 15-3, 15-4, 16-9, 19-7, 19-8, ... 21-1, 21-2, 22-1, 22-6, 25-3
CSF$ 16-6,22-4, 22-5, 22-6
DA 6-10
DAN 6-10, 6-11
data structures 1-2, 2-7, 9-6, 20-1, 20-3
data table 4-6, 9-2
DATE$ 11-9,22-1,22-2,22-3
debugging 15-5, 17-2, 18-3, 20-10, 22-4, 23-3, 25-4, 25-5, 25-6
DEC 6-6, 10-1, 10-3, 10-4, 21-5
DEC2 ..... 6-6, 10-3, 10-4
DEF 16-7, 16-8, 16-13
DI 6-8, 6-9, 7-6, 11-7, 22-3, 22-4
DJZ 8-2
DL 6-10, 6-11, 9-10
DLM 6-10
DLN 6-10
DLSC 11-7
double-linked list 20-2, 20-7, 20-10
DS 6-10, 6-11

DSA 11-7
DSC 11-7
DSL 11-7, 11-9, 22-3, 22-4
DTE 8-7
EABT$ 12-2, 16-13, 25-7
ERR$ 12-2, 16-13
error flags 4-3, 5-5, 5-6, 5-7, 16-5, 23-1, 25-1
error type 25-2
error termination 25-3, 25-7
error code 25-2, 25-3
ERs ..... iii, 11-9, 12-1, 12-2, 12-3, 12-5, 13-1, 13-5, 13-6, 13-9, 13-10, 13-12, 16-13, 17-2, ... 19-1, 19-10, 22-5
EX 14-1, 14-4, 21-3, 21-4, 21-5
EXIT$ 12-2, 14-7, 16-13, 21-6
external definitions 2-5, 2-6
external references 2-5, 2-6, 5-6, 23-1
f 3-9, 5-6, 6-2, 6-7, 13-9, 21-1, 25-5
Fieldata ..... 3-7, 3-9, 4-7, 4-9, 4-10, 4-11, 6-12, 9-11, 11-7, 11-9, 12-2, 12-4, 13-1, 13-4, ... 13-5, 13-6, 13-7, 13-9, 13-10, 13-13, 13-14, 14-7, 15-7, 16-12, 17-4, 19-5, 21-1, 21-3, ... 21-4, 21-5, 21-7, 21-8, 22-1, 22-2, 22-4, 25-5
fixed-point 3-6, 4-7, 4-10, 6-10, 6-11, 17-5
FLIT 1-2, 25-7, 25-8
floating point 2-2, 3-6, 3-7, 4-7, 17-1, 25-5
FRSTIN 23-2, 23-3
GRS 7-2, 14-6, 15-2
h 3-9, 3-10, 5-6, 6-2, 7-7, 9-6, 9-7, 9-8, 10-2, 14-2, 15-1, 15-3, 21-6, 25-5
H 13-11, 13-12
heading 13-10, 13-11, 13-13
immediate operands 7-6, 9-7
IN 23-2
INC 6-7, 10-1, 10-3, 10-4
INC2 ..... 6-7, 10-3, 10-4
increment field 3-10,9-6
index register designator 3-10
index registers 7-6, 9-8, 9-10, 15-1, 19-3, 21-4, 21-6, 25-4
index register incrementation 3-10, 7-7, 9-6, 9-7, 15-4
indexing 3-2, 3-10, 6-9, 9-1, 9-2, 9-7
indirect addressing 9-8, 9-9
INFOR 13-3
instruction word ..... 3-9, 4-2, 5-7, 6-2, 6-6, 6-7, 7-6, 7-7, 8-9, 9-6, 9-8, 9-11, 11-8, 11-9, 11-10, ... 19-7, 19-8, 25-5
IOAU 2-1
IOP 2-1
IOU 2-1, 3-2
IP 2-1, 3-1, 3-2, 21-1
j 3-9, 5-6, 6-2, 6-10, 7-2, 7-3, 7-6, 8-1, 10-3, 15-3, 16-10, 20-5, 21-1, 25-5
J ..... 8-1, 8-2, 8-3, 8-5, 8-6, 8-7, 8-8, 8-9, 8-11, 9-2, 9-3, 9-4, 9-5, 9-7, 9-8, 9-12, 10-2, 10-4, ... 10-5, 11-4, 11-6, 12-4, 13-3, 14-2, 14-3, 14-4, 15-3, 15-5, 15-6, 18-3, 18-4, 18-5, 19-6, ... 19-7, 20-6, 20-9, 21-3, 21-4, 21-5, 21-7, 21-8, 21-9
JB 8-4
JGD 10-1,10-3,10-5,19-4, 21-3, 21-7, 21-8, 21-9

JN 8-3, 13-2, 21-7, 21-8, 21-9
JNB 8-4
JNZ 8-2, 8-3, 8-8, 8-9, 11-4,20-9, 22-6
JP 8-3, 13-2
jump history stack 25-3
JZ 8-2, 8-3, 8-9, 20-6
L$SNAP 12-3, 12-4, 12-5, 13-14, 16-12, 18-4, 21-6, 25-6
LA 4-2, 6-1, 6-2, 6-3, 6-5, 6-10
label ..... 4-1, 4-2, 4-3, 4-6, 4-7, 4-8, 4-9, 4-10, 4-11, 5-6, 6-9, 7-5, 7-6, 8-1, 8-2, 8-10, 9-11, ... 12-5, 13-7, 14-1, 14-4, 16-1, 16-2, 16-3, 16-4, 16-5, 16-7, 16-8, 16-11, 16-13, 18-1, 18-2, ... 19-4, 19-5, 19-6, 19-7, 20-3, 21-1, 21-5, 22-2, 25-6
LBJ 14-4
LDJ 14-4, 14-5
LDSC 11-7, 21-7, 21-8, 21-9
LDSL 11-7
LIB 23-2, 23-3
library .....
LIJ 3-2, 14-4, 25-2
linked list deletion 20-6
linked list insertion 20-6
linked lists 20-2, 20-3, 20-5, 20-6, 20-10, 20-11
linked list sorting 20-7
list header 20-2, 20-4
list trailer 20-2
listing format 5-5
literals 9-10, 9-11, 16-8, 19-5
LMJ 14-2, 14-3, 14-4, 14-7, 16-10, 19-3, 20-9, 21-3, 21-4, 21-7, 25-2, 25-4
LN 6-3, 6-4, 8-6
LNMA 6-4
location counter ..... 2-6, 4-1, 4-2, 4-3, 4-4, 4-5, 4-8, 4-11, 5-2, 5-6, 6-1, 6-2, 9-10, 16-1, 16-5, ... 16-8, 16-9, 16-12, 16-13, 18-1, 22-2, 23-2, 25-3, 25-4
logical shifts 11-1
logical operators 17-2, 17-4, 17-5
LPD 7-4, 21-2
LR 6-2, 6-3, 6-5
LRS 14-5, 14-6
LSC 11-7, 11-10
LSSC 11-3
LSSL 11-2, 11-8, 21-7, 21-8, 21-9
LX 6-2, 6-3, 6-5, 6-9, 9-8
LXI 9-7, 9-8, 10-2, 21-6
LXM 9-8, 21-3, 21-4
M$SNAP 16-12,
margins 5-5, 13-10, 13-11, 13-13, 13-14
MAXR$ 4-4, 4-5
MI 6-8, 6-9
MLU 11-3, 11-6, 11-7, 11-10
MSE 15-5, 15-6
MSG 15-6,
MSI 6-8, 6-9, 7-6, 8-8, 21-7, 21-8, 21-9
MSLE 15-5

MSNE 15-5
NI 6-6, 8-5, 8-8, 8-9, 8-11, 9-2, 10-2, 10-3, 11-6, 14-2, 14-3, 15-2, 15-4, 15-5, 19-7
node layout 20-5
NOT 5-4, 23-3
octal arithmetic 3-3, 3-4, 3-5, 3-6
octal conversion 3-5, 3-6
operation field 4-2, 4-4, 4-5, 4-6, 4-7, 16-2, 16-4, 18-1, 18-2
OPT$ 5-3, 22-2
OR 3-2, 11-4, 11-6
P$RINT 13-5, 13-10, 16-2, 16-3, 18-3, 18-4, 18-5
P$RNTA 13-8, 13-10
partial words 4-5
partial word ..... 3-9, 4-2, 4-4, 4-5, 6-7, 6-10, 7-2, 7-3, 7-5, 8-4, 9-4, 9-7, 9-11, 11-6,11-9, 15-1, ... 15-2, 15-3 15-5, 16-10, 19-3, 21-1, 22-2
pipelining 19-7
PMD 3-6, 4-4, 12-1, 12-4, 19-5, 25-3, 25-4, 25-5, 25-8
precision control 17-5
PRINT$ ..... 5-4, 9-11, 11-8, 11-9, 12-1, 13-1, 13-4, 13-5, 13-6, 13-8, 13-9, 13-10, 13-12, 16-2, ... 16-3, 17-4, 22-1, 24-1
PRINTA$ 12-1, 13-8
PRTCN$ 13-10, 13-11, 13-13
PSR$ 21-1, 21-2
quarter-word designator 7-4
quarter-word mode 7-4, 7-5, 7-7, 21-1, 21-2, 25-3
queue 13-8, 20-10
R$EAD 13-3, 13-10
R$EADA 13-7, 13-9, 13-10
READ$ ..... iii, 12-1, 13-1, 13-2, 13-3, 13-4, 13-5, 13-6, 13-10, 13014, 21-7, 22-1, 24-1
READA$ 12-1, 13-6, 13-7, 13-8, 13-10
reference manuals ..... iii, 1-1, 1-3, 2-7, 3-10, 22-6
register designator 3-10
registers overlap 6-9
relational operators 17-4, 17-5
SA 6-1, 6-2, 6-3, 6-5, 6-10
SAS 6-7
SAZ 6-7
SE 15-2, 15-3, 15-4, 15-5, 15-7
SFS 6-7, 7-6
SFZ 6-7
SG 15-3
signed character strings 4-10
single-linked list 20-2
SLE 15-2
SLJ 9-9, 14-1, 14-2
SN 6-3, 6-4
SN1 6-7
SNAP$ 12-2, 12-3, 12-4, 12-5, 13-14, 16-12, 25-6
SNE 15-2
SNG 15-2
SNW 15-4
SNZ 6-7

SP1 6-7
SPD 21-2
special forms request 13-12
SR 6-2, 6-3, 6-5
SRS 14-5, 14-6
SSC 11-3
SSL 11-1, 11-2, 11-8, 11-9, 22-3, 22-4
stack 20-10, 23-1, 25-3
string operators 17-4
string functions 4-10
style ..... iv, 3-8, 4-2, 4-3, 4-4, 4-6, 4-8, 4-10, 8-1, 8-2, 8-10, 8-11, 9-3, 9-8, 12-4, 13-1, 14-1, ... 14-3, 15-1, 17-1, 19-1
subassembly 4-8, 16-1, 16-4, 16-5, 16-8
SW 15-4
SX 6-2, 6-3, 6-5
symbols 2-5, 4-1, 4-7, 7-11, 5-6, 25-1
SZ 6-7, 7-6, 8-5, 8-6, 9-2, 9-3, 9-4, 9-5, 9-7, 10-2, 11-4, 20-9
TDATE$ 22-3
TE 4-10, 8-7, 8-8, 14-4, 15-5, 19-6
TEP 11-6
TG 8-7, 8-9, 19-7, 20-9, 21-7, 21-8, 21-9, 25-7
TIME$ 22-4
TLE 8-7, 8-9, 8-10, 9-2, 9-3, 9-4, 9-5, 9-6,, 9-7, 10-2
TN 8-5, 8-6
TNE 8-7, 8-8, 8-9, 16-13, 19-7, 20-6, 21-7, 21-8, 21-9
TNW 8-10, 8-11
TNZ 8-5, 8-7, 8-9, 10-5, 11-4, 18-5, 20-6, 20-9
TOP 11-6, 13-3
TP 8-5, 8-6
traversing a linked list 20-5, 20-6
TW 8-10, 8-11, 15-4
TYPE QUARTER 21-1
TYPE THIRD 21-1
TZ 8-5, 9-2, 9-3, 9-4, 9-5, 9-6, 9-7, 9-8, 10-2, 11-4, 20-9
u 3-9, 3-10, 5-6, 7-5, 12-1, 25-2, 25-5
U ..... 3-10, 4-10, 6-2, 6-8, 6-10, 6-11, 7-6, 8-3, 8-7, 8-8, 8-9, 8-10, 8-11, 9-1, 9-2, 9-3, ... 9-4, 9-5, 9-6, 9-7, 9-8, 9-9, 9-11, 10-2, 10-3, 11-1, 11-2, 11-3, 11-4, 11-5, 11-6, ... 11-7, 11-8, 11-9, 12-3, 13-6, 13-8, 14-1, 14-2, 14-4, 14-5, 15-2, 15-3, 15-4, 15-5, ... 16-6, 16-12, 19-7, 19-8, 19-9, 20-6, 21-1, 21-2, 21-3, 21-4, 21-6, 21-7, 21-8, 21-9, ... 22-3, 22-4, 22-5, 25-6, 25-7
waiting label 4-3, 16-1, 16-4, 16-5, 16-13, 19-6, 21-5, 22-2
x 3-9, 3-10
XI 9-1, 9-6, 11-3, 14-4, 15-1
XM 9-1, 9-6, 9-7, 9-8, 11-2, 11-3, 14-4, 22-3
XOR 3-2, 11-3, 11-4, 11-5, 17-3

FOR THE USERS of the two thousand Unisys (formerly Sperry) Series 1100 and 2200 systems all over the world, there are few if any non-Unisys materials available to assist them in programming. With this book, the author leads the reader through the intricacies of the 1100/2200 MASM environment. The aim here is to offer help in learning CPU features, in learning MASM processor features and directives and in identifying key information sources. Moreover, the author gives code templates for using many instructions and ERs and offers a style guide for assembly language.

Michael Maddox joined the Univac Division of Sperry Rand in July of 1973 in an assignment at Bell Laboratories. Since then, he has had assignments at some of Sperry and Unisys' largest customers, including the Internal Revenue Service, the U.S. Department of Agriculture and the Texas Department of Public Welfare (now Department of Human Services). He has done a variety of programming, ranging from systems software development (on BICS-1100 at Bell Laboratories) and systems software support at several sites, to transaction program development at other sites. He is now working at Glazier Electronics in Fairfield, Iowa.

ISBN 0-9627241-0-6